# Ghana Electric Vehicle Grid Integration Analysis (Enhanced Complete)

# Comprehensive Techno-Economic Analysis of EV Impact on National Grid

# Added: Seasonal Analysis, Weekend/Weekday Patterns, Reserve Margin Analysis

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

import seaborn as sns

from datetime import datetime, timedelta

import warnings

from sklearn.preprocessing import StandardScaler

from sklearn.cluster import KMeans

from scipy import stats

import plotly.graph\_objects as go

from plotly.subplots import make\_subplots

import plotly.express as px

# Configure display options

pd.set\_option('display.max\_columns', None)

pd.set\_option('display.width', None)

warnings.filterwarnings('ignore')

# Set plotting style

plt.style.use('seaborn-v0\_8')

sns.set\_palette("husl")

print("=== GHANA EV GRID INTEGRATION ANALYSIS (ENHANCED COMPLETE) ===")

print("Optimizing Electric Vehicle Integration in Ghana's Power Grid")

print("A Comprehensive Techno-Economic Analysis with Seasonal & Reserve Analysis\n")

#=============================================================================

# 1. DATA LOADING AND PREPROCESSING (Enhanced)

#=============================================================================

def load\_and\_prepare\_data(file\_path):

"""

Load and preprocess the Ghana hourly electricity demand data

Enhanced with seasonal and temporal features

"""

print("📊 Loading Ghana Hourly Electricity Demand Data...")

try:

# Load the data

df = pd.read\_excel(file\_path)

# Display basic info about the dataset

print(f"✅ Data loaded successfully!")

print(f"📈 Dataset shape: {df.shape}")

print(f"📅 Date range: {df['Timestamp'].min()} to {df['Timestamp'].max()}")

# Convert timestamp to datetime

df['Timestamp'] = pd.to\_datetime(df['Timestamp'])

# Create additional time-based features

df['Hour'] = df['Timestamp'].dt.hour

df['Day'] = df['Timestamp'].dt.day

df['Month'] = df['Timestamp'].dt.month

df['DayOfWeek'] = df['Timestamp'].dt.dayofweek

df['DayName'] = df['Timestamp'].dt.day\_name()

df['MonthName'] = df['Timestamp'].dt.month\_name()

df['IsWeekend'] = df['DayOfWeek'].isin([5, 6])

# Enhanced seasonal features for Ghana's climate

df['Season'] = df['Month'].map({

12: 'Dry', 1: 'Dry', 2: 'Dry',

3: 'Hot\_Dry', 4: 'Hot\_Dry', 5: 'Hot\_Dry',

6: 'Rainy', 7: 'Rainy', 8: 'Rainy',

9: 'Post\_Rainy', 10: 'Post\_Rainy', 11: 'Post\_Rainy'

})

df['Quarter'] = df['Timestamp'].dt.quarter

df['WeekOfYear'] = df['Timestamp'].dt.isocalendar().week

# Calculate additional metrics

df['EV\_Penetration\_Percent'] = (df['EV Load Only (MW)'] / df['Base Load (MW)']) \* 100

df['Total\_Load\_Increase\_Percent'] = ((df['With EV Load (MW)'] - df['Base Load (MW)']) / df['Base Load (MW)']) \* 100

# Ghana grid capacity assumption (estimated for West African countries)

df['Grid\_Capacity\_MW'] = 5000 # Assumed grid capacity

df['Base\_Reserve\_Margin'] = ((df['Grid\_Capacity\_MW'] - df['Base Load (MW)']) / df['Base Load (MW)']) \* 100

df['EV\_Reserve\_Margin'] = ((df['Grid\_Capacity\_MW'] - df['With EV Load (MW)']) / df['With EV Load (MW)']) \* 100

print(f"🔧 Enhanced features created successfully!")

return df

except Exception as e:

print(f"❌ Error loading data: {str(e)}")

return None

# Load the data

file\_path = r"C:\Users\USER\Desktop\Homer Project\Ghana\_2024\_Hourly\_Load\_with\_EV.xlsx"

df = load\_and\_prepare\_data(file\_path)

#=============================================================================

# 2. SEASONAL EV IMPACT ANALYSIS

#=============================================================================

class SeasonalEVAnalyzer:

"""

Comprehensive seasonal EV impact analysis

"""

def \_\_init\_\_(self, data):

self.data = data

self.results = {}

def analyze\_seasonal\_variations(self):

"""Analyze seasonal variations in EV charging patterns"""

print("\n🌦️ ANALYZING SEASONAL EV IMPACT VARIATIONS...")

# Seasonal summary statistics

seasonal\_stats = self.data.groupby('Season').agg({

'Base Load (MW)': ['mean', 'max', 'std'],

'With EV Load (MW)': ['mean', 'max', 'std'],

'EV Load Only (MW)': ['mean', 'max', 'std'],

'EV\_Penetration\_Percent': ['mean', 'max', 'std'],

'Base\_Reserve\_Margin': ['mean', 'min'],

'EV\_Reserve\_Margin': ['mean', 'min']

}).round(2)

# Hourly patterns by season

seasonal\_hourly = self.data.groupby(['Season', 'Hour']).agg({

'Base Load (MW)': 'mean',

'With EV Load (MW)': 'mean',

'EV Load Only (MW)': 'mean'

}).reset\_index()

# Critical comparison: January vs July analysis

jan\_data = self.data[self.data['Month'] == 1] # Dry season

jul\_data = self.data[self.data['Month'] == 7] # Rainy season

jan\_peak\_ev = jan\_data['EV Load Only (MW)'].max()

jul\_peak\_ev = jul\_data['EV Load Only (MW)'].max()

jan\_peak\_total = jan\_data['With EV Load (MW)'].max()

jul\_peak\_total = jul\_data['With EV Load (MW)'].max()

# AC demand impact analysis (higher in hot seasons)

seasonal\_increase = ((jul\_peak\_total - jan\_peak\_total) / jan\_peak\_total) \* 100

self.results['seasonal\_stats'] = seasonal\_stats

self.results['seasonal\_hourly'] = seasonal\_hourly

self.results['jan\_vs\_jul\_comparison'] = {

'jan\_peak\_ev': jan\_peak\_ev,

'jul\_peak\_ev': jul\_peak\_ev,

'ev\_seasonal\_increase': ((jul\_peak\_ev - jan\_peak\_ev) / jan\_peak\_ev) \* 100,

'total\_load\_seasonal\_increase': seasonal\_increase

}

print(f"🌧️ July Peak EV Load: {jul\_peak\_ev:.1f} MW")

print(f"❄️ January Peak EV Load: {jan\_peak\_ev:.1f} MW")

print(f"📈 Seasonal EV Load Increase: {self.results['jan\_vs\_jul\_comparison']['ev\_seasonal\_increase']:.1f}%")

print(f"🌡️ Total Load Seasonal Increase: {seasonal\_increase:.1f}%")

return self.results

def identify\_seasonal\_super\_peaks(self):

"""Identify when AC demand + EV charging creates summer super-peaks"""

print("\n🚨 IDENTIFYING SEASONAL SUPER-PEAK PERIODS...")

# Define super-peak threshold (top 1% of loads)

super\_peak\_threshold = self.data['With EV Load (MW)'].quantile(0.99)

super\_peaks = self.data[self.data['With EV Load (MW)'] >= super\_peak\_threshold]

# Analyze super-peaks by season and hour

super\_peak\_seasons = super\_peaks.groupby('Season').size()

super\_peak\_hours = super\_peaks.groupby('Hour').size()

super\_peak\_months = super\_peaks.groupby('MonthName').size().sort\_values(ascending=False)

# Calculate EV contribution during super-peaks

super\_peak\_ev\_contribution = super\_peaks['EV Load Only (MW)'].mean()

normal\_ev\_contribution = self.data['EV Load Only (MW)'].mean()

ev\_amplification\_factor = super\_peak\_ev\_contribution / normal\_ev\_contribution

# Summer super-peak analysis (Hot\_Dry season)

summer\_super\_peaks = super\_peaks[super\_peaks['Season'] == 'Hot\_Dry']

summer\_amplification = len(summer\_super\_peaks) / len(super\_peaks) \* 100

self.results['super\_peaks'] = {

'threshold\_mw': super\_peak\_threshold,

'count': len(super\_peaks),

'by\_season': super\_peak\_seasons,

'by\_hour': super\_peak\_hours,

'by\_month': super\_peak\_months,

'ev\_amplification\_factor': ev\_amplification\_factor,

'summer\_concentration': summer\_amplification,

'super\_peak\_data': super\_peaks

}

print(f"⚡ Super-Peak Threshold: {super\_peak\_threshold:.1f} MW")

print(f"🎯 Number of Super-Peak Periods: {len(super\_peaks)}")

print(f"🔋 EV Amplification Factor during Super-Peaks: {ev\_amplification\_factor:.2f}x")

print(f"☀️ Summer Super-Peak Concentration: {summer\_amplification:.1f}%")

print(f"🌡️ Most Critical Month: {super\_peak\_months.index[0]} ({super\_peak\_months.iloc[0]} occurrences)")

return super\_peaks

#=============================================================================

# 3. WEEKEND VS WEEKDAY EV BEHAVIOR ANALYSIS

#=============================================================================

class WeekendWeekdayAnalyzer:

"""

Comprehensive weekend vs weekday EV behavior analysis

"""

def \_\_init\_\_(self, data):

self.data = data

self.results = {}

def analyze\_weekend\_weekday\_patterns(self):

"""Analyze weekend vs weekday EV charging patterns"""

print("\n📅 ANALYZING WEEKEND VS WEEKDAY EV BEHAVIOR...")

# Separate weekend and weekday data

weekday\_data = self.data[~self.data['IsWeekend']]

weekend\_data = self.data[self.data['IsWeekend']]

# Hourly patterns comparison

weekday\_hourly = weekday\_data.groupby('Hour').agg({

'Base Load (MW)': 'mean',

'With EV Load (MW)': 'mean',

'EV Load Only (MW)': 'mean',

'EV\_Penetration\_Percent': 'mean'

})

weekend\_hourly = weekend\_data.groupby('Hour').agg({

'Base Load (MW)': 'mean',

'With EV Load (MW)': 'mean',

'EV Load Only (MW)': 'mean',

'EV\_Penetration\_Percent': 'mean'

})

# Peak analysis

weekday\_ev\_peak\_hour = weekday\_hourly['EV Load Only (MW)'].idxmax()

weekend\_ev\_peak\_hour = weekend\_hourly['EV Load Only (MW)'].idxmax()

weekday\_ev\_peak\_load = weekday\_hourly.loc[weekday\_ev\_peak\_hour, 'EV Load Only (MW)']

weekend\_ev\_peak\_load = weekend\_hourly.loc[weekend\_ev\_peak\_hour, 'EV Load Only (MW)']

# Statistical comparison

ev\_load\_ttest = stats.ttest\_ind(weekday\_data['EV Load Only (MW)'],

weekend\_data['EV Load Only (MW)'])

# Behavioral insights

behavioral\_metrics = {

'weekday\_peak\_hour': weekday\_ev\_peak\_hour,

'weekend\_peak\_hour': weekend\_ev\_peak\_hour,

'peak\_hour\_shift': weekend\_ev\_peak\_hour - weekday\_ev\_peak\_hour,

'weekday\_peak\_load': weekday\_ev\_peak\_load,

'weekend\_peak\_load': weekend\_ev\_peak\_load,

'weekend\_vs\_weekday\_ratio': weekend\_ev\_peak\_load / weekday\_ev\_peak\_load,

'statistical\_significance': ev\_load\_ttest.pvalue < 0.05,

'p\_value': ev\_load\_ttest.pvalue

}

self.results['weekday\_hourly'] = weekday\_hourly

self.results['weekend\_hourly'] = weekend\_hourly

self.results['behavioral\_metrics'] = behavioral\_metrics

print(f"🏢 Weekday EV Peak: Hour {weekday\_ev\_peak\_hour} ({weekday\_ev\_peak\_load:.1f} MW)")

print(f"🏠 Weekend EV Peak: Hour {weekend\_ev\_peak\_hour} ({weekend\_ev\_peak\_load:.1f} MW)")

print(f"⏰ Peak Hour Shift: {behavioral\_metrics['peak\_hour\_shift']} hours")

print(f"📊 Weekend/Weekday Ratio: {behavioral\_metrics['weekend\_vs\_weekday\_ratio']:.2f}")

return self.results

def identify\_weekend\_grid\_stress(self):

"""Identify different grid stress patterns for weekends vs weekdays"""

print("\n⚡ ANALYZING WEEKEND VS WEEKDAY GRID STRESS PATTERNS...")

# Calculate stress scores separately for weekends and weekdays

weekday\_data = self.data[~self.data['IsWeekend']].copy()

weekend\_data = self.data[self.data['IsWeekend']].copy()

# Weekend-specific stress calculation

weekend\_data['Weekend\_Stress\_Score'] = (

(weekend\_data['With EV Load (MW)'] / weekend\_data['With EV Load (MW)'].max()) \* 0.4 +

(weekend\_data['EV Load Only (MW)'] / weekend\_data['EV Load Only (MW)'].max()) \* 0.4 +

(weekend\_data['EV\_Penetration\_Percent'] / weekend\_data['EV\_Penetration\_Percent'].max()) \* 0.2

)

# Weekday-specific stress calculation

weekday\_data['Weekday\_Stress\_Score'] = (

(weekday\_data['With EV Load (MW)'] / weekday\_data['With EV Load (MW)'].max()) \* 0.4 +

(weekday\_data['EV Load Only (MW)'] / weekday\_data['EV Load Only (MW)'].max()) \* 0.4 +

(weekday\_data['EV\_Penetration\_Percent'] / weekday\_data['EV\_Penetration\_Percent'].max()) \* 0.2

)

# Critical periods identification

weekend\_critical = weekend\_data[weekend\_data['Weekend\_Stress\_Score'] >=

weekend\_data['Weekend\_Stress\_Score'].quantile(0.9)]

weekday\_critical = weekday\_data[weekday\_data['Weekday\_Stress\_Score'] >=

weekday\_data['Weekday\_Stress\_Score'].quantile(0.9)]

stress\_comparison = {

'weekend\_critical\_hours': weekend\_critical.groupby('Hour').size(),

'weekday\_critical\_hours': weekday\_critical.groupby('Hour').size(),

'weekend\_avg\_stress': weekend\_data['Weekend\_Stress\_Score'].mean(),

'weekday\_avg\_stress': weekday\_data['Weekday\_Stress\_Score'].mean(),

'weekend\_max\_stress': weekend\_data['Weekend\_Stress\_Score'].max(),

'weekday\_max\_stress': weekday\_data['Weekday\_Stress\_Score'].max()

}

self.results['stress\_comparison'] = stress\_comparison

print(f"📈 Weekend Average Stress: {stress\_comparison['weekend\_avg\_stress']:.3f}")

print(f"📈 Weekday Average Stress: {stress\_comparison['weekday\_avg\_stress']:.3f}")

print(f"🚨 Weekend Critical Periods: {len(weekend\_critical)}")

print(f"🚨 Weekday Critical Periods: {len(weekday\_critical)}")

return stress\_comparison

#=============================================================================

# 4. GRID RESERVE MARGIN ANALYSIS

#=============================================================================

class GridReserveAnalyzer:

"""

Comprehensive grid reserve margin and capacity analysis

"""

def \_\_init\_\_(self, data, grid\_capacity\_mw=5000):

self.data = data

self.grid\_capacity = grid\_capacity\_mw

self.results = {}

def calculate\_reserve\_margins(self):

"""Calculate comprehensive reserve margin analysis"""

print(f"\n🔋 CALCULATING GRID RESERVE MARGINS (Capacity: {self.grid\_capacity} MW)...")

# Basic reserve margin calculations

base\_min\_reserve = self.data['Base\_Reserve\_Margin'].min()

ev\_min\_reserve = self.data['EV\_Reserve\_Margin'].min()

# Critical reserve periods (< 10% reserve)

critical\_reserve\_threshold = 10

base\_critical\_periods = self.data[self.data['Base\_Reserve\_Margin'] < critical\_reserve\_threshold]

ev\_critical\_periods = self.data[self.data['EV\_Reserve\_Margin'] < critical\_reserve\_threshold]

# Emergency periods (< 5% reserve)

emergency\_threshold = 5

base\_emergency = self.data[self.data['Base\_Reserve\_Margin'] < emergency\_threshold]

ev\_emergency = self.data[self.data['EV\_Reserve\_Margin'] < emergency\_threshold]

# Load shedding requirement analysis

load\_shedding\_periods = self.data[self.data['With EV Load (MW)'] > self.grid\_capacity]

reserve\_analysis = {

'base\_min\_reserve': base\_min\_reserve,

'ev\_min\_reserve': ev\_min\_reserve,

'reserve\_degradation': base\_min\_reserve - ev\_min\_reserve,

'critical\_periods\_base': len(base\_critical\_periods),

'critical\_periods\_ev': len(ev\_critical\_periods),

'emergency\_periods\_base': len(base\_emergency),

'emergency\_periods\_ev': len(ev\_emergency),

'load\_shedding\_periods': len(load\_shedding\_periods),

'max\_load\_shedding\_required': max(0, self.data['With EV Load (MW)'].max() - self.grid\_capacity)

}

self.results['reserve\_analysis'] = reserve\_analysis

print(f"⚠️ Minimum Base Reserve Margin: {base\_min\_reserve:.1f}%")

print(f"🚨 Minimum EV Reserve Margin: {ev\_min\_reserve:.1f}%")

print(f"📉 Reserve Degradation: {reserve\_analysis['reserve\_degradation']:.1f}%")

print(f"🔴 Critical Periods (Base): {reserve\_analysis['critical\_periods\_base']}")

print(f"🔴 Critical Periods (With EV): {reserve\_analysis['critical\_periods\_ev']}")

print(f"⚡ Load Shedding Required: {reserve\_analysis['load\_shedding\_periods']} periods")

return reserve\_analysis

def identify\_capacity\_limit\_periods(self):

"""Identify specific hours/days requiring load shedding"""

print("\n🚨 IDENTIFYING CAPACITY LIMIT PERIODS...")

# Periods approaching capacity limits (>90% utilization)

high\_utilization\_threshold = 0.9 \* self.grid\_capacity

high\_util\_periods = self.data[self.data['With EV Load (MW)'] > high\_utilization\_threshold]

# Overload periods (>100% capacity)

overload\_periods = self.data[self.data['With EV Load (MW)'] > self.grid\_capacity]

if len(overload\_periods) > 0:

# Analyze overload patterns

overload\_hours = overload\_periods.groupby('Hour').size().sort\_values(ascending=False)

overload\_days = overload\_periods.groupby('DayName').size().sort\_values(ascending=False)

overload\_months = overload\_periods.groupby('MonthName').size().sort\_values(ascending=False)

# Calculate required load shedding

overload\_periods['Load\_Shedding\_Required'] = overload\_periods['With EV Load (MW)'] - self.grid\_capacity

overload\_periods['EV\_Shedding\_Potential'] = overload\_periods['EV Load Only (MW)'] \* 0.7

capacity\_results = {

'high\_util\_periods': len(high\_util\_periods),

'overload\_periods': len(overload\_periods),

'overload\_hours': overload\_hours,

'overload\_days': overload\_days,

'overload\_months': overload\_months,

'max\_overload': overload\_periods['Load\_Shedding\_Required'].max(),

'avg\_overload': overload\_periods['Load\_Shedding\_Required'].mean(),

'overload\_data': overload\_periods

}

print(f"⚡ High Utilization Periods: {len(high\_util\_periods)}")

print(f"🚨 Overload Periods: {len(overload\_periods)}")

print(f"📊 Maximum Overload: {capacity\_results['max\_overload']:.1f} MW")

print(f"🕐 Most Critical Hour: {overload\_hours.index[0]} ({overload\_hours.iloc[0]} occurrences)")

else:

capacity\_results = {

'high\_util\_periods': len(high\_util\_periods),

'overload\_periods': 0,

'overload\_message': 'No capacity overload detected'

}

print(f"✅ No capacity overload detected")

print(f"⚠️ High utilization periods: {len(high\_util\_periods)}")

self.results['capacity\_analysis'] = capacity\_results

return capacity\_results

#=============================================================================

# 5. ENHANCED VISUALIZATION SUITE

#=============================================================================

class EnhancedVisualizationSuite:

"""Enhanced visualization suite"""

def \_\_init\_\_(self, data, seasonal\_results, weekend\_results, reserve\_results):

self.data = data

self.seasonal\_results = seasonal\_results

self.weekend\_results = weekend\_results

self.reserve\_results = reserve\_results

def create\_seasonal\_analysis\_dashboard(self):

"""Create seasonal analysis dashboard"""

fig, axes = plt.subplots(2, 3, figsize=(24, 12))

fig.suptitle('Seasonal EV Impact Analysis - Ghana Grid Integration', fontsize=16, fontweight='bold')

# 1. Seasonal load comparison

seasonal\_summary = self.data.groupby('Season')[['Base Load (MW)', 'With EV Load (MW)', 'EV Load Only (MW)']].mean()

x = np.arange(len(seasonal\_summary.index))

width = 0.25

axes[0,0].bar(x - width, seasonal\_summary['Base Load (MW)'], width, label='Base Load', alpha=0.8)

axes[0,0].bar(x, seasonal\_summary['With EV Load (MW)'], width, label='With EV Load', alpha=0.8)

axes[0,0].bar(x + width, seasonal\_summary['EV Load Only (MW)'], width, label='EV Load Only', alpha=0.8)

axes[0,0].set\_title('Average Load by Season')

axes[0,0].set\_xlabel('Season')

axes[0,0].set\_ylabel('Load (MW)')

axes[0,0].set\_xticks(x)

axes[0,0].set\_xticklabels(seasonal\_summary.index, rotation=45)

axes[0,0].legend()

axes[0,0].grid(True, alpha=0.3)

# 2. Jan vs Jul comparison

jan\_data = self.data[self.data['Month'] == 1]

jul\_data = self.data[self.data['Month'] == 7]

jan\_hourly = jan\_data.groupby('Hour')['EV Load Only (MW)'].mean()

jul\_hourly = jul\_data.groupby('Hour')['EV Load Only (MW)'].mean()

axes[0,1].plot(jan\_hourly.index, jan\_hourly.values, 'b-', linewidth=3, label='January', marker='o')

axes[0,1].plot(jul\_hourly.index, jul\_hourly.values, 'r-', linewidth=3, label='July', marker='s')

axes[0,1].set\_title('January vs July EV Charging Patterns')

axes[0,1].set\_xlabel('Hour of Day')

axes[0,1].set\_ylabel('EV Load (MW)')

axes[0,1].legend()

axes[0,1].grid(True, alpha=0.3)

# 3. Super-peak analysis

if 'super\_peaks' in self.seasonal\_results:

super\_peak\_seasons = self.seasonal\_results['super\_peaks']['by\_season']

axes[0,2].bar(super\_peak\_seasons.index, super\_peak\_seasons.values, color='red', alpha=0.7)

axes[0,2].set\_title('Super-Peak Occurrences by Season')

axes[0,2].set\_xlabel('Season')

axes[0,2].set\_ylabel('Number of Super-Peaks')

axes[0,2].grid(True, alpha=0.3)

# 4. Reserve margin by season

seasonal\_reserves = self.data.groupby('Season')[['Base\_Reserve\_Margin', 'EV\_Reserve\_Margin']].mean()

x = np.arange(len(seasonal\_reserves.index))

axes[1,0].bar(x - 0.2, seasonal\_reserves['Base\_Reserve\_Margin'], 0.4, label='Base Grid', alpha=0.8)

axes[1,0].bar(x + 0.2, seasonal\_reserves['EV\_Reserve\_Margin'], 0.4, label='With EV', alpha=0.8)

axes[1,0].set\_title('Reserve Margin by Season')

axes[1,0].set\_xlabel('Season')

axes[1,0].set\_ylabel('Reserve Margin (%)')

axes[1,0].set\_xticks(x)

axes[1,0].set\_xticklabels(seasonal\_reserves.index, rotation=45)

axes[1,0].legend()

axes[1,0].grid(True, alpha=0.3)

# 5. Weekend vs Weekday comparison

weekday\_hourly = self.weekend\_results['weekday\_hourly']

weekend\_hourly = self.weekend\_results['weekend\_hourly']

axes[1,1].plot(weekday\_hourly.index, weekday\_hourly['EV Load Only (MW)'], 'b-',

linewidth=3, label='Weekday', marker='o')

axes[1,1].plot(weekend\_hourly.index, weekend\_hourly['EV Load Only (MW)'], 'r-',

linewidth=3, label='Weekend', marker='s')

axes[1,1].set\_title('Weekend vs Weekday EV Patterns')

axes[1,1].set\_xlabel('Hour of Day')

axes[1,1].set\_ylabel('EV Load (MW)')

axes[1,1].legend()

axes[1,1].grid(True, alpha=0.3)

# 6. Grid capacity utilization

capacity\_util = (self.data['With EV Load (MW)'] / self.data['Grid\_Capacity\_MW'] \* 100)

monthly\_util = self.data.groupby('Month').apply(lambda x: (x['With EV Load (MW)'] / x['Grid\_Capacity\_MW'] \* 100).max())

axes[1,2].plot(monthly\_util.index, monthly\_util.values, 'g-', linewidth=3, marker='o', markersize=6)

axes[1,2].axhline(y=90, color='orange', linestyle='--', label='High Utilization (90%)')

axes[1,2].axhline(y=100, color='red', linestyle='--', label='Capacity Limit (100%)')

axes[1,2].set\_title('Monthly Peak Grid Utilization')

axes[1,2].set\_xlabel('Month')

axes[1,2].set\_ylabel('Grid Utilization (%)')

axes[1,2].legend()

axes[1,2].grid(True, alpha=0.3)

plt.tight\_layout()

plt.show()

def create\_reserve\_margin\_dashboard(self):

"""Create reserve margin analysis dashboard"""

fig, axes = plt.subplots(2, 2, figsize=(20, 12))

fig.suptitle('Grid Reserve Margin & Capacity Analysis', fontsize=16, fontweight='bold')

# 1. Reserve margin time series

sample\_data = self.data.sample(min(1000, len(self.data))).sort\_values('Timestamp')

axes[0,0].plot(sample\_data['Timestamp'], sample\_data['Base\_Reserve\_Margin'],

'b-', alpha=0.7, label='Base Grid')

axes[0,0].plot(sample\_data['Timestamp'], sample\_data['EV\_Reserve\_Margin'],

'r-', alpha=0.7, label='With EV')

axes[0,0].axhline(y=10, color='orange', linestyle='--', label='Critical (10%)')

axes[0,0].axhline(y=5, color='red', linestyle='--', label='Emergency (5%)')

axes[0,0].set\_title('Reserve Margin Over Time')

axes[0,0].set\_xlabel('Time')

axes[0,0].set\_ylabel('Reserve Margin (%)')

axes[0,0].legend()

axes[0,0].grid(True, alpha=0.3)

# 2. Load vs Capacity

####################

# 2. Load vs Capacity

axes[0,1].scatter(self.data['With EV Load (MW)'], self.data['Grid\_Capacity\_MW'],

alpha=0.6, c=self.data['Hour'], cmap='viridis', s=30)

axes[0,1].plot([0, self.data['Grid\_Capacity\_MW'].iloc[0]],

[0, self.data['Grid\_Capacity\_MW'].iloc[0]],

'r--', linewidth=2, label='Capacity Limit')

axes[0,1].axvline(x=self.data['Grid\_Capacity\_MW'].iloc[0] \* 0.9,

color='orange', linestyle='--', label='90% Capacity')

axes[0,1].set\_title('Load vs Grid Capacity')

axes[0,1].set\_xlabel('Load with EV (MW)')

axes[0,1].set\_ylabel('Grid Capacity (MW)')

axes[0,1].legend()

axes[0,1].grid(True, alpha=0.3)

# 3. Critical periods heatmap

hourly\_reserves = self.data.pivot\_table(

values='EV\_Reserve\_Margin',

index='Hour',

columns='Month',

aggfunc='mean'

)

im = axes[1,0].imshow(hourly\_reserves.values, cmap='RdYlGn', aspect='auto')

axes[1,0].set\_title('Reserve Margin Heatmap (Hour vs Month)')

axes[1,0].set\_xlabel('Month')

axes[1,0].set\_ylabel('Hour of Day')

axes[1,0].set\_xticks(range(12))

axes[1,0].set\_xticklabels(range(1, 13))

axes[1,0].set\_yticks(range(0, 24, 4))

axes[1,0].set\_yticklabels(range(0, 24, 4))

plt.colorbar(im, ax=axes[1,0], label='Reserve Margin (%)')

# 4. Capacity utilization distribution

capacity\_util = (self.data['With EV Load (MW)'] / self.data['Grid\_Capacity\_MW'] \* 100)

axes[1,1].hist(capacity\_util, bins=50, alpha=0.7, color='skyblue', edgecolor='black')

axes[1,1].axvline(x=90, color='orange', linestyle='--', linewidth=2, label='High Utilization')

axes[1,1].axvline(x=100, color='red', linestyle='--', linewidth=2, label='Overload')

axes[1,1].set\_title('Grid Capacity Utilization Distribution')

axes[1,1].set\_xlabel('Capacity Utilization (%)')

axes[1,1].set\_ylabel('Frequency')

axes[1,1].legend()

axes[1,1].grid(True, alpha=0.3)

plt.tight\_layout()

plt.show()

def create\_interactive\_seasonal\_dashboard(self):

"""Create interactive Plotly dashboard for seasonal analysis"""

print("\n📊 Creating Interactive Seasonal Analysis Dashboard...")

# Create subplot structure

fig = make\_subplots(

rows=3, cols=2,

subplot\_titles=('Seasonal EV Load Patterns', 'Jan vs Jul Hourly Comparison',

'Super-Peak Analysis by Season', 'Weekend vs Weekday Patterns',

'Reserve Margin by Season', 'Capacity Utilization Trends'),

specs=[[{"secondary\_y": False}, {"secondary\_y": False}],

[{"secondary\_y": False}, {"secondary\_y": False}],

[{"secondary\_y": False}, {"secondary\_y": False}]]

)

# 1. Seasonal patterns

for season in self.data['Season'].unique():

season\_data = self.data[self.data['Season'] == season]

hourly\_pattern = season\_data.groupby('Hour')['EV Load Only (MW)'].mean()

fig.add\_trace(

go.Scatter(x=hourly\_pattern.index, y=hourly\_pattern.values,

mode='lines+markers', name=f'{season} Season',

line=dict(width=3)),

row=1, col=1

)

# 2. Jan vs Jul comparison

jan\_data = self.data[self.data['Month'] == 1].groupby('Hour')['EV Load Only (MW)'].mean()

jul\_data = self.data[self.data['Month'] == 7].groupby('Hour')['EV Load Only (MW)'].mean()

fig.add\_trace(

go.Scatter(x=jan\_data.index, y=jan\_data.values,

mode='lines+markers', name='January (Dry)',

line=dict(color='blue', width=3)),

row=1, col=2

)

fig.add\_trace(

go.Scatter(x=jul\_data.index, y=jul\_data.values,

mode='lines+markers', name='July (Rainy)',

line=dict(color='red', width=3)),

row=1, col=2

)

# 3. Super-peak analysis

if 'super\_peaks' in self.seasonal\_results:

super\_peak\_data = self.seasonal\_results['super\_peaks']['by\_season']

fig.add\_trace(

go.Bar(x=super\_peak\_data.index, y=super\_peak\_data.values,

name='Super-Peaks', marker\_color='crimson'),

row=2, col=1

)

# 4. Weekend vs Weekday

weekday\_hourly = self.weekend\_results['weekday\_hourly']['EV Load Only (MW)']

weekend\_hourly = self.weekend\_results['weekend\_hourly']['EV Load Only (MW)']

fig.add\_trace(

go.Scatter(x=weekday\_hourly.index, y=weekday\_hourly.values,

mode='lines+markers', name='Weekday',

line=dict(color='blue', width=3)),

row=2, col=2

)

fig.add\_trace(

go.Scatter(x=weekend\_hourly.index, y=weekend\_hourly.values,

mode='lines+markers', name='Weekend',

line=dict(color='orange', width=3)),

row=2, col=2

)

# 5. Reserve margins by season

seasonal\_reserves = self.data.groupby('Season')[['Base\_Reserve\_Margin', 'EV\_Reserve\_Margin']].mean()

fig.add\_trace(

go.Bar(x=seasonal\_reserves.index, y=seasonal\_reserves['Base\_Reserve\_Margin'],

name='Base Grid Reserve', marker\_color='lightblue'),

row=3, col=1

)

fig.add\_trace(

go.Bar(x=seasonal\_reserves.index, y=seasonal\_reserves['EV\_Reserve\_Margin'],

name='With EV Reserve', marker\_color='darkblue'),

row=3, col=1

)

# 6. Capacity utilization trends

monthly\_util = self.data.groupby('Month').apply(

lambda x: (x['With EV Load (MW)'] / x['Grid\_Capacity\_MW'] \* 100).max()

)

fig.add\_trace(

go.Scatter(x=monthly\_util.index, y=monthly\_util.values,

mode='lines+markers', name='Peak Utilization',

line=dict(color='green', width=4)),

row=3, col=2

)

# Add critical thresholds

fig.add\_hline(y=90, line\_dash="dash", line\_color="orange", row=3, col=2)

fig.add\_hline(y=100, line\_dash="dash", line\_color="red", row=3, col=2)

# Update layout

fig.update\_layout(

height=1200,

title\_text="Ghana EV Grid Integration - Seasonal & Behavioral Analysis Dashboard",

title\_font\_size=20,

showlegend=True

)

# Update axis labels

fig.update\_xaxes(title\_text="Hour of Day", row=1, col=1)

fig.update\_xaxes(title\_text="Hour of Day", row=1, col=2)

fig.update\_xaxes(title\_text="Season", row=2, col=1)

fig.update\_xaxes(title\_text="Hour of Day", row=2, col=2)

fig.update\_xaxes(title\_text="Season", row=3, col=1)

fig.update\_xaxes(title\_text="Month", row=3, col=2)

fig.update\_yaxes(title\_text="EV Load (MW)", row=1, col=1)

fig.update\_yaxes(title\_text="EV Load (MW)", row=1, col=2)

fig.update\_yaxes(title\_text="Super-Peak Count", row=2, col=1)

fig.update\_yaxes(title\_text="EV Load (MW)", row=2, col=2)

fig.update\_yaxes(title\_text="Reserve Margin (%)", row=3, col=1)

fig.update\_yaxes(title\_text="Utilization (%)", row=3, col=2)

fig.show()

#=============================================================================

# 6. COMPREHENSIVE ANALYSIS EXECUTION

#=============================================================================

def run\_comprehensive\_analysis():

"""Execute the complete enhanced analysis"""

if df is None:

print("❌ Data loading failed. Cannot proceed with analysis.")

return

print("\n" + "="\*80)

print("🚀 EXECUTING COMPREHENSIVE EV GRID INTEGRATION ANALYSIS")

print("="\*80)

# Initialize analyzers

seasonal\_analyzer = SeasonalEVAnalyzer(df)

weekend\_analyzer = WeekendWeekdayAnalyzer(df)

reserve\_analyzer = GridReserveAnalyzer(df, grid\_capacity\_mw=5000)

# Execute seasonal analysis

print("\n🌦️ Phase 1: Seasonal Analysis")

seasonal\_results = seasonal\_analyzer.analyze\_seasonal\_variations()

super\_peaks = seasonal\_analyzer.identify\_seasonal\_super\_peaks()

# Execute weekend/weekday analysis

print("\n📅 Phase 2: Weekend vs Weekday Analysis")

weekend\_results = weekend\_analyzer.analyze\_weekend\_weekday\_patterns()

stress\_results = weekend\_analyzer.identify\_weekend\_grid\_stress()

# Execute reserve margin analysis

print("\n🔋 Phase 3: Grid Reserve Margin Analysis")

reserve\_results = reserve\_analyzer.calculate\_reserve\_margins()

capacity\_results = reserve\_analyzer.identify\_capacity\_limit\_periods()

# Create visualizations

print("\n📊 Phase 4: Creating Enhanced Visualizations")

viz\_suite = EnhancedVisualizationSuite(df, seasonal\_results, weekend\_results, reserve\_results)

# Static dashboard

viz\_suite.create\_seasonal\_analysis\_dashboard()

viz\_suite.create\_reserve\_margin\_dashboard()

# Interactive dashboard

viz\_suite.create\_interactive\_seasonal\_dashboard()

return {

'seasonal\_results': seasonal\_results,

'weekend\_results': weekend\_results,

'reserve\_results': reserve\_results,

'capacity\_results': capacity\_results

}

#=============================================================================

# 7. CRITICAL INSIGHTS AND RECOMMENDATIONS GENERATOR

#=============================================================================

class CriticalInsightsGenerator:

"""Generate actionable insights and recommendations"""

def \_\_init\_\_(self, data, analysis\_results):

self.data = data

self.results = analysis\_results

def generate\_seasonal\_insights(self):

"""Generate seasonal-specific insights"""

print("\n🌦️ SEASONAL INSIGHTS & RECOMMENDATIONS")

print("="\*60)

seasonal\_results = self.results['seasonal\_results']

# Peak season identification

if 'jan\_vs\_jul\_comparison' in seasonal\_results:

comparison = seasonal\_results['jan\_vs\_jul\_comparison']

seasonal\_increase = comparison['ev\_seasonal\_increase']

print(f"🔥 SEASONAL PEAK AMPLIFICATION:")

print(f" • EV demand increases {seasonal\_increase:.1f}% from January to July")

print(f" • July peak EV load: {comparison['jul\_peak\_ev']:.1f} MW")

print(f" • January peak EV load: {comparison['jan\_peak\_ev']:.1f} MW")

if seasonal\_increase > 20:

print(" ⚠️ CRITICAL: Seasonal variation >20% requires adaptive charging strategies")

print(" 💡 RECOMMENDATION: Implement seasonal EV charging tariffs")

print(" 💡 RECOMMENDATION: Deploy seasonal demand response programs")

# Super-peak analysis

if 'super\_peaks' in seasonal\_results:

super\_peak\_data = seasonal\_results['super\_peaks']

summer\_concentration = super\_peak\_data['summer\_concentration']

print(f"\n⚡ SUPER-PEAK CONCENTRATION:")

print(f" • {summer\_concentration:.1f}% of super-peaks occur in hot season")

print(f" • EV amplification factor: {super\_peak\_data['ev\_amplification\_factor']:.2f}x")

if summer\_concentration > 50:

print(" 🚨 CRITICAL: Summer super-peak clustering detected")

print(" 💡 RECOMMENDATION: Deploy energy storage for summer peak shaving")

print(" 💡 RECOMMENDATION: Implement smart charging with seasonal algorithms")

def generate\_behavioral\_insights(self):

"""Generate weekend/weekday behavioral insights"""

print("\n📅 BEHAVIORAL INSIGHTS & RECOMMENDATIONS")

print("="\*60)

weekend\_results = self.results['weekend\_results']

behavioral\_metrics = weekend\_results['behavioral\_metrics']

peak\_shift = behavioral\_metrics['peak\_hour\_shift']

weekend\_ratio = behavioral\_metrics['weekend\_vs\_weekday\_ratio']

print(f"🕐 CHARGING BEHAVIOR PATTERNS:")

print(f" • Peak hour shifts {peak\_shift} hours on weekends")

print(f" • Weekend peak is {weekend\_ratio:.2f}x weekday peak")

if abs(peak\_shift) >= 2:

print(" ⚠️ SIGNIFICANT: Weekend charging behavior differs substantially")

print(" 💡 RECOMMENDATION: Implement time-differentiated weekend tariffs")

if weekend\_ratio > 1.2:

print(" 📈 Higher weekend EV usage detected")

print(" 💡 RECOMMENDATION: Weekend-specific grid reinforcement planning")

elif weekend\_ratio < 0.8:

print(" 📉 Lower weekend EV usage - opportunity for load shifting")

print(" 💡 RECOMMENDATION: Promote weekend charging incentives")

def generate\_capacity\_insights(self):

"""Generate grid capacity and reserve insights"""

print("\n🔋 GRID CAPACITY INSIGHTS & RECOMMENDATIONS")

print("="\*60)

reserve\_results = self.results['reserve\_results']

capacity\_results = self.results['capacity\_results']

min\_reserve = reserve\_results['ev\_min\_reserve']

reserve\_degradation = reserve\_results['reserve\_degradation']

print(f"📊 RESERVE MARGIN ANALYSIS:")

print(f" • Minimum reserve margin with EV: {min\_reserve:.1f}%")

print(f" • Reserve degradation due to EV: {reserve\_degradation:.1f}%")

if min\_reserve < 10:

print(" 🚨 CRITICAL: Reserve margin below 10% threshold")

print(" 💡 URGENT: Implement emergency demand response protocols")

print(" 💡 URGENT: Consider grid capacity expansion")

if min\_reserve < 5:

print(" ⛔ EMERGENCY: Reserve margin below 5% - load shedding risk")

print(" 💡 IMMEDIATE: Deploy automatic load shedding for EV charging")

# Overload analysis

if 'overload\_periods' in capacity\_results and capacity\_results['overload\_periods'] > 0:

overload\_count = capacity\_results['overload\_periods']

max\_overload = capacity\_results['max\_overload']

print(f"\n⚡ CAPACITY OVERLOAD DETECTED:")

print(f" • {overload\_count} periods require load shedding")

print(f" • Maximum overload: {max\_overload:.1f} MW")

print(" 🚨 CRITICAL: Grid capacity insufficient for full EV integration")

print(" 💡 IMMEDIATE: Implement smart charging with grid constraints")

print(" 💡 STRATEGIC: Plan grid capacity expansion")

def generate\_comprehensive\_recommendations(self):

"""Generate comprehensive strategic recommendations"""

print("\n🎯 COMPREHENSIVE STRATEGIC RECOMMENDATIONS")

print("="\*70)

print("1️⃣ IMMEDIATE ACTIONS (0-6 months):")

print(" • Deploy smart charging infrastructure with grid awareness")

print(" • Implement time-of-use tariffs for EV charging")

print(" • Establish emergency load shedding protocols for EV charging")

print(" • Install real-time grid monitoring systems")

print("\n2️⃣ SHORT-TERM STRATEGIES (6-18 months):")

print(" • Develop seasonal EV charging algorithms")

print(" • Deploy distributed energy storage systems")

print(" • Implement vehicle-to-grid (V2G) pilot programs")

print(" • Create weekend-specific charging incentives")

print("\n3️⃣ MEDIUM-TERM PLANNING (1-3 years):")

print(" • Expand grid capacity for identified bottlenecks")

print(" • Deploy community-scale energy storage")

print(" • Implement comprehensive demand response programs")

print(" • Develop EV-solar integration strategies")

print("\n4️⃣ LONG-TERM VISION (3-5 years):")

print(" • Achieve 100% renewable-powered EV charging")

print(" • Implement fully automated grid management")

print(" • Deploy nationwide V2G infrastructure")

print(" • Establish EV charging as grid stabilization service")

# Economic analysis

total\_ev\_energy = (df['EV Load Only (MW)'] \* 1).sum() # Approximate daily energy

ev\_charging\_cost = total\_ev\_energy \* 0.12 # Assumed cost per kWh

print(f"\n💰 ECONOMIC IMPACT SUMMARY:")

print(f" • Estimated daily EV energy consumption: {total\_ev\_energy:.0f} MWh")

print(f" • Estimated daily charging revenue: ${ev\_charging\_cost:.0f}")

print(f" • Annual EV charging revenue potential: ${ev\_charging\_cost \* 365:.0f}")

print("\n📋 KEY PERFORMANCE INDICATORS TO MONITOR:")

print(" • Reserve margin maintenance above 10%")

print(" • Zero unplanned load shedding events")

print(" • EV charging cost competitiveness vs conventional vehicles")

print(" • Grid stability metrics during peak charging periods")

print(" • Customer satisfaction with charging availability")

# Execute the comprehensive analysis

if \_\_name\_\_ == "\_\_main\_\_":

# Run the complete analysis

analysis\_results = run\_comprehensive\_analysis()

if analysis\_results:

# Generate insights and recommendations

print("\n" + "="\*80)

print("🎯 GENERATING CRITICAL INSIGHTS AND RECOMMENDATIONS")

print("="\*80)

insights\_generator = CriticalInsightsGenerator(df, analysis\_results)

insights\_generator.generate\_seasonal\_insights()

insights\_generator.generate\_behavioral\_insights()

insights\_generator.generate\_capacity\_insights()

insights\_generator.generate\_comprehensive\_recommendations()

print("\n" + "="\*80)

print("✅ GHANA EV GRID INTEGRATION ANALYSIS COMPLETED")

print("📊 All visualizations, insights, and recommendations generated")

print("🎯 Ready for strategic implementation planning")

print("="\*80)